Efficient Decentralized Monitoring of Safety in Distributed Systems

Koushik Sen, Abhay Vardhan, Gul Agha, Grigore Roşu
Department of Computer Science
University of Illinois at Urbana Champaign
{ksen,vardhan,agha,grosu}@cs.uiuc.edu

Abstract

We describe an efficient decentralized monitoring algorithm that monitors a distributed program's execution to check for violations of safety properties. The monitoring is based on formulae written in PT-DTL*, a variant of past time linear temporal logic that we define.* PT-DTL *is suitable for expressing temporal properties of distributed systems. Specifically, the formulae of* PT-DTL *are relative to a particular process and are interpreted over a projection of the trace of global states that represents what that process is aware of. A formula relative to one process may refer to other processes' local states through remote expressions and remote formulae. In order to correctly evaluate remote expressions, we introduce the notion of* KNOWLEDGEVECTOR *and provide an algorithm which keeps a process aware of other processes' local states that can affect the validity of a monitored* PT-DTL *formula. Both the logic and the monitoring algorithm are illustrated through a number of examples. Finally, we describe our implementation of the algorithm in a tool called* DIANA.

1. Introduction

Software errors from a number of different problems such as incorrect or incomplete specifications, coding errors, and faults and failures in the hardware, operating system or network. Model checking is an important technology which is finding increasing use as a means of reducing software errors. Unfortunately, despite impressive recent advances, the size of systems for which model checking is feasible remains rather limited. This weakness is particularly critical in the context of distributed systems: concurrency and asynchrony results in inherent non-determinism that significantly increases the number of states to be analyzed. As a result, most system builders must continue to use testing to identify bugs in their implementations.

There are two problems with software testing. First, testing is generally done in an *ad hoc* manner: the software developer must hand translate the requirements into specific dynamic checks on the program state. Second, test coverage is often rather limited, covering only some execution paths. To mitigate the first problem, software often includes dynamic checks on a system's state in order to identify problems at run-time. Recently, there has been some interest in run-time monitoring techniques which provide a little more rigor in testing. In this approach, monitors are automatically synthesized from a formal specification. These monitors may then be deployed off-line for debugging or on-line for dynamically checking that safety properties are not being violated during system execution.

In this paper, we argue that distributed systems may be effectively monitored at runtime against formally specified safety requirements. By effective monitoring, we mean not only linear efficiency, but also decentralized monitoring where few or no additional messages need to be passed for monitoring purposes. We introduce an epistemic temporal logic for distributed knowledge. We illustrate the expressiveness of this logic by means of some simple examples. We then show how efficient distributed monitors may be synthesized from the specified requirements. Finally, we describe a distributed systems application development framework, called DIANA. To use DIANA, a user must provide an application together with the formal safety properties that she wants monitored. DIANA automatically synthesizes code for monitoring the specified requirements and weaves appropriate instrumentation code into the given application. The architecture of DIANA is illustrated in Figure 1.

The work presented in this paper was stimulated by the observation that in many distributed systems, such as wireless sensor networks, it is quite impractical to monitor requirements expressed in classical temporal logics. For example, consider a system of mobile nodes in which one mobile node may request a certain value from another mobile node. On receiving the request, the second node computes the value and returns it. An important requirement in such a system is that no node receives a reply from a node to which it has not previously issued a request. It is easy to see that Linear Temporal Logic (LTL) would not be a practical specification language for any reasonably sized collec-

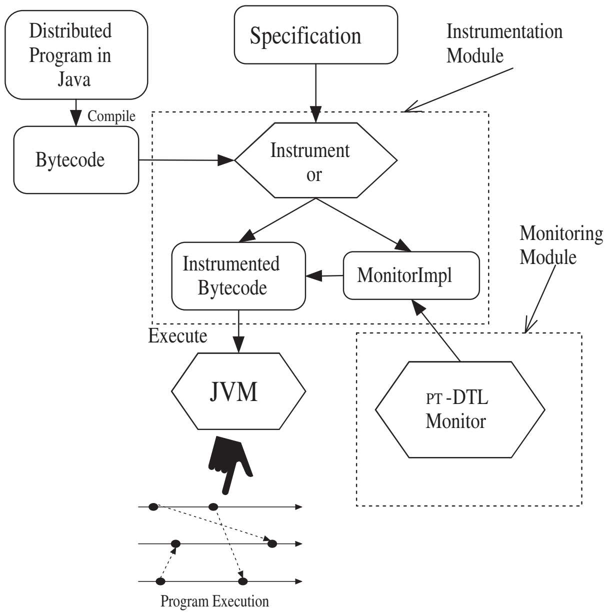

Figure 1. The Architecture of DIANA

tion of nodes. To use LTL, we would need to collect consistent snapshots of the global system; a monitor would then check the snapshots for possible violations of the property by considering all possible interleavings of events that are allowed by the distributed computation. In a system of thousands of nodes, collecting such a global snapshot would be prohibitive. Moreover, the number of possible interleavings to be considered would be large even if powerful techniques such as partial order reduction are used.

To address the above difficulty, we define *past-time distributed temporal logic* (PT-DTL). Using PT-DTL, one can check a property such as the one above by having a local monitor on each node. For example, node a monitors "if a has received a value then it must be the case that previously in the past at b the following held: b has computed the value and at a a request was made for that value in the past". This is precisely and concisely expressed by the PT-DTL formula:

$$receivedValue \rightarrow \\ @_b(\Diamond(computedValue \wedge @_a(\Diamond requestedValue)))$$

Note that we read $@$ as "at", $@_b F$ is the value of F in the most recent local state of b that the current process is aware of, and $\Diamond$ denotes the formula was true sometime in the past. Monitoring the above formula involves sending no additional messages – it involves inserting only a few bits of information which are piggybacked on the messages that are already being passed in the computation. This efficiency provides a substantial improvement over what is required to monitor formulas written in classical LTL.

We introduce *remote expressions* in PT-DTL to represent values which are functions depending on the state of a remote process. For example, a process may monitor the property: "if my alarm has been set then it must be the case that the difference between my temperature and the temperature at process b exceeded the allowed value". This is expressed as:

$$alarm \rightarrow \Diamond((myTemp - @_b otherTemp) > allowed)$$

Here $@_b otherTemp$ is a remote expression that is subtracted from the local value of $myTemp$.

An example of a safety property that may be useful in the context of an airplane software is: "if my airplane is landing then the runway allocated by the airport matches the one that I am planning to use". This property may be expressed in PT-DTL as follows:

$$landing \rightarrow (runway = (@_{airport} allocRunway))$$

Many researchers have proposed temporal logics to reason about distributed systems. Most of these logics are inspired by the classic work of Aumann [5] and Halpern *et al.* [7] on knowledge in distributed systems. Meenakshi *et al.* define a knowledge temporal logic interpreted over a message sequence charts in a distributed system [16] and develop methods for model checking formulae in this logic. Our communication primitive was in part inspired by this work, but we allow arbitrary expressions and atomic propositions over expressions in their logic.

Another closely related work is that of Penczek [17, 18] which defines a temporal logic of causal knowledge. Knowledge operators are provided to reason about the local history of a process, as well as about the knowledge it acquires from other processes. However, in order to keep the complexity of model checking tractable, Penczek does not allow the nesting of causal knowledge operators. Interestingly, the nesting of causal knowledge operators does not add any complexity to our algorithm for monitoring.

Leucker investigates linear temporal logic interpreted over restricted labeled partial orders called Mazurkiewicz traces [12]. An overview of distributed linear time temporal logics based on Mazurkiewicz traces is given by Thiagaranjan *et al.* in [22]. Alur *et al.* [4] introduce a temporal logic of causality (TLC) which is interpreted over causal structures corresponding to partial order executions of a distributed system. They use both past and future time operators and give a model checking algorithm for the logic.

In recent years, there has been considerable interest in runtime verification [1]. Havelund *et al.* [10] give algorithms for synthesizing efficient monitors for safety properties. Sen *et al.* [20] develop techniques for runtime safety analysis for multithreaded programs and introduce the tool JMPAX. Some other runtime verification systems include JPaX from NASA Ames [9] and UPENN's Mac [11].

We can think of at least three major contributions of the work presented in this paper. First, we define a simple but expressive logic to specify safety properties in distributed systems. Second, we provide an algorithm to synthesize decentralized monitors for safety properties that are expressed in the logic. Finally, we describe the implementation of a tool (DIANA) that is based on this technique. The tool is publicly available for download.

The rest of the paper is organized as follows. Section 2 and Section 3 give the preliminaries. Section 4 introduces PT-DTL. In Section 5 we describe the algorithm that underlies our implementation. Section 6 briefly describes the implementation along with initial experimentation.

2. Distributed Systems

A distributed system is a collection of n processes or actors $(p_1, \ldots, p_n)$, each with its own local state. The local state of a process is given by the values bound to its variables. Note that there are no global or shared variables. Processes communicate with other using asynchronous messages whose order of arrival is indeterminate. The computation of each process is abstractly modeled by a set of *events*, and a distributed computation is specified by a partial order $\prec$ on the events. There are three types of events:

1. *internal* events change the local state of a process;
2. *send* events cause a process to send a message; and
3. *receive* events occur when a message is received by a process.

Let E_i denote the set of events of process p_i and let E denote $\bigcup_i E_i$. Now, $\lessdot \subseteq E \times E$ is defined as follows:

1. $e \lessdot e'$ if e and e' are events of the same process and e happens immediately before e',
2. $e \lessdot e'$ if e is the send event of a message at some process and e' is the corresponding receive event of the message at the recipient process.

The partial order $\prec$ is the transitive closure of the relation $\lessdot$. This partial order captures the *causality* relation between events. The structure described by $\mathcal{C} = (E, \prec)$ is called a *distributed computation* and we assume an arbitrary but given distributed computation $\mathcal{C}$. Further, $\preccurlyeq$ is the reflexive and transitive closure of $\lessdot$. In Fig. 2, $e_{11} \prec e_{23}$, $e_{12} \prec e_{23}$, and $e_{11} \lessdot e_{23}$. However, $e_{12} \not\lessdot e_{23}$.

For $e \in E$, we define $\downarrow e \stackrel{\text{def}}{=} \{e' \mid e' \preccurlyeq e\}$, that is, $\downarrow e$ is the set of events that causally precede e. For $e \in E_i$, we can think of $\downarrow e$ as the local state of p_i when the event e has just occurred. This state contains the history of events of all processes that causally precede e.

We extend the definition of $\lessdot$, $\prec$ and $\preccurlyeq$ to local states such that $\downarrow e \lessdot \downarrow e'$ iff $e \lessdot e'$, $\downarrow e \prec \downarrow e'$ iff $e \prec e'$, and $\downarrow e \preccurlyeq \downarrow e'$ iff $e \preccurlyeq e'$. We denote the set of local states of a process p_i by $LS_i \stackrel{\text{def}}{=} \{\downarrow e \mid e \in E_i\}$ and let $LS \stackrel{\text{def}}{=} \bigcup_i LS_i$. We use the symbols $s_i, s_i', s_i'', \ldots$ to represent the local states of process p_i. We also assume that the local state s_i of each process p_i associates values to some local variables V_i, and that $s_i(v)$ denotes the value of a variable $v \in V_i$ in the local state s_i at process p_i.

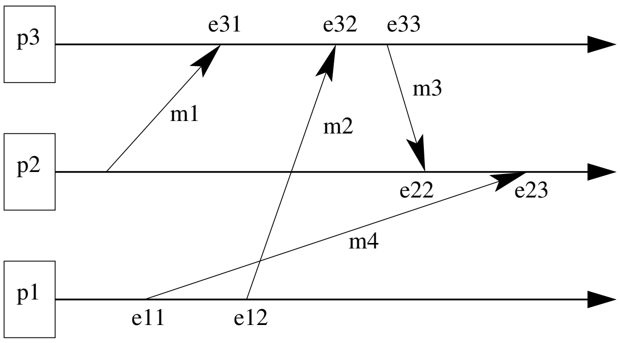

Figure 2. Sample Distributed Computation

We use the notation $causal_j(s_i)$ to refer to the latest state of process p_j that the process p_i knows while in state s_i. Formally, if $causal_j(s_i) = s_j$ then $s_j \in LS_j$ and $s_j \preccurlyeq s_i$ and for all $s_j' \in LS_j$ if $s_j' \preccurlyeq s_i$ then $s_j' \preccurlyeq s_j$. For example, in Figure 2 $causal_1(\downarrow e_{23}) = \downarrow e_{12}$. Note that if $i = j$ then $causal_j(s_i) = s_i$.

3. Past Time Linear Temporal Logic (PT-LTL)

Past-time Linear Temporal Logic (PT-LTL) [13, 14] has been used in [10, 11, 20] to express, monitor and predict violations of safety properties of software systems. The syntax of PT-LTL is as follows:

$$\begin{array}{lll} F ::= & \text{true} \mid \text{false} \mid a \in A \mid \neg F \mid F \; op \; F & \text{propositional} \\ & \mid \odot F \mid \Diamond F \mid \boxdot F \mid F \,\mathcal{S}\, F & \text{temporal} \end{array}$$

where *op* are standard binary operators, $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$. $\odot F$ should be read as "previously", $\Diamond F$ as "eventually in the past", $\boxdot F$ as "always in the past", $F_1 \mathcal{S} F_2$ as "F_1 since F_2".

The logic is interpreted on a finite sequence of states or a *run*. If $\rho = s_1 s_2 \ldots s_n$ is a run then we let ρ_i denote the prefix run $s_1 s_2 \ldots s_i$ for each $1 \leq i \leq n$. The semantics of the different operators is given in Table 1.

For example, the formula $\boxdot((action \wedge \odot\neg action) \rightarrow (\neg stop \;\mathcal{S}\; start))$ states that whenever *action* starts to be true, it is the case that *start* was true at some point in the past and since then *stop* was never true: in other words, the action is taken only while the system is active.

Notice that the semantics of "previously" is given as if the trace is unbounded in the past and stationary in the first event. In runtime monitoring, we start the process of monitoring from the point that the first event is generated and we continue monitoring for as long as events are generated.

$\rho \models \text{true}$	for all ρ,
$\rho \not\models \text{false}$	for all ρ,
$\rho \models a$	iff a holds in the state s_n,
$\rho \models \neg F$	iff $\rho \not\models F$,
$\rho \models F_1 \; op \; F_2$	iff $\rho \models F_1$ and/or/implies/iff $\rho \models F_2$, when op is $\wedge / \vee / \rightarrow / \leftrightarrow$,
$\rho \models \odot F$	iff $\rho' \models F$, where $\rho' = \rho_{n-1}$ if $n > 1$ and $\rho' = \rho$ if $n = 1$,
$\rho \models \Diamond F$	iff $\rho_i \models F$ for some $1 \leq i \leq n$,
$\rho \models \boxdot F$	iff $\rho_i \models F$ for all $1 \leq i \leq n$,
$\rho \models F_1 \; \mathcal{S} \; F_2$	iff $\rho_j \models F_2$ for some $1 \leq j \leq n$ and $\rho_i \models F_1$ for all $j < i \leq n$,

Table 1. Semantics of PT-LTL

Although PT-LTL is interpreted over a linear execution trace, in distributed systems a computation is a partial order which may have several possible linearizations. Therefore, monitoring a distributed computation requires monitoring all possible linear traces that may be obtained from a partial order. Unfortunately, the number of linearizations of a partial order may be exponential in the length of the computation and thus monitoring PT-LTL formula may be intractable. A major contribution of this paper is to extend PT-LTL so that we can reason about a distributed property using only local monitoring. We describe this extension next.

4. Past Time Distributed Temporal Logic

Although PT-LTL works well for a single process, once we have more processes interacting with each other we need to reason about the state of remote processes. Since practical distributed systems are usually asynchronous and the absolute global state of the system is *not* available to processes, the best thing that each process can do it to reason about the global state that it is *is aware of.*

We define Past-Time Distributed Temporal Logic (PT-DTL) by extending PT-LTL to express safety properties of distributed message passing systems. Specifically, we add a pair of *epistemic operators* as in [19], written @, whose role is to evaluate an expression or a formula in the *last known state* of a remote process. We call such an expression or a formula *remote*. A remote expression or formula may contain nested epistemic operators and refer to variables that are local to a remote process. By using remote expressions, in addition to remote formulae, a larger class of desirable properties of distributed systems may be specified without sacrificing the efficiency of monitoring.

For example, consider the simple local property at a process p_i that if α is true in the current local state of p_i then β must be true at the latest state of process p_j of which p_i is aware of. This property will be written formally in PT-DTL as $\alpha \rightarrow @_j\beta$. However, referring to remote formulae only is *not* sufficient to express a broad range of useful global properties such as "at process p_i, the value of x in the current state is greater than the value of y at process p_j in the latest causally preceding state." The reason we introduce the novel epistemic operators on expressions is that it is crucial to be able to also refer to *values* of expressions in remote local states. For example, the property above can be formally specified as the PT-DTL formula $x > @_j y$ at process p_i where $@_j y$ is the value of y at process p_j that p_i is aware of.

The intuition underlying PT-DTL is that each process is associated with local temporal formulae which may refer to the global state of the distributed system. These formulae are required to be valid at the respective processes during a distributed computation. A distributed computation satisfies the specification when all the local formulae are shown to satisfy the computation.

4.1. Syntax

From now on, we will use PT-DTL formula only in the context of a particular process, say p_i. We call such formulae *i-formulae* and denote them as $F_i, F_i', \ldots$. Moreover, we introduce *i-expressions*, expressions that are local to a process p_i, and denoted them by $\xi_i, \xi_i', \ldots$. Informally, an i-expression is an expression over the global state of the system that process p_i is currently aware of. Local predicates on i-expressions form the atomic propositions on which the temporal i-formulae are built.

We add the *epistemic operators* $@_j$ that take j-expressions or j-formulae and convert them into expressions or formulae local to process p_i. Informally, $@_j$ yields an expression or a formula on process p_j over the projection of the global state that the current process is aware of. The following gives the formal syntax of PT-DTL with respect to a process p_i, where i and j are any process indices (not necessarily distinct):

F_i	$::=$	$\text{true} \mid \text{false} \mid P(\vec{\xi_i}) \mid \neg F_i \mid F_i \; op \; F_i$	propositional
		$\mid \odot F_i \mid \Diamond F_i \mid \boxdot F_i \mid F_i \; \mathcal{S} \; F_i$	temporal
		$\mid @_j F_j$	epistemic
ξ_i	$::=$	$c \mid v_i \mid f(\vec{\xi_i})$	functional
		$\mid @_j \xi_j$	epistemic
$\vec{\xi_i}$	$::=$	$(\xi_i, \ldots, \xi_i)$	

The infix operator *op* may be a binary propositional operator such as $\wedge, \vee, \rightarrow$ or $\equiv$. The term $\vec{\xi_i}$ stands for a tuple of expressions on process p_i. The term $P(\vec{\xi_i})$ is a (computable) predicate over the tuple $\vec{\xi_i}$ and $f(\vec{\xi_i})$ is a (computable) function over the tuple. For example, P may be $<, \leq, >, \geq, =$ and f may be $+, -, /, *$. Variables v_i belongs to the set V_i which contains all the local state variables of process p_i. Constants such as $0, 1, 3.4$ are represented by $c, c', c_1, \ldots$.

The expression $@_j\xi_j$ is an i-expression representing the remote expression ξ_j. Similarly, $@_jF_j$ is an i-formula referring to the local knowledge about the remote validity of j-formula F_j. In other words, $@_j$ converts a j-expression or a j-formula to an i-expression or an i-formula, respectively.

4.2. Semantics

The semantics of PT-DTL is a natural extension of PT-LTL with the expected behavior for the epistemic operators. The atomic propositions of PT-LTL are replaced by predicates over tuples of expressions. Table 2 formally gives the semantics of each operator of PT-DTL. $(\mathcal{C}, s_i)[\![@_j\xi_j]\!]$ is the value of the expression ξ_j in the state $s_j = causal(s_i)$ which is the latest state of process p_j of which process p_i is aware of. We assume that expressions are properly typed. Typically, these types could be: `integer, real, strings`. We assume that $s_i, s_i', s_i'', \ldots \in LS_i$ and $s_j, s_j', s_j'', \ldots \in LS_j$. Notice that, as in PT-LTL, the meaning of the "previously" operator on the initial state of each process reflects the intuition that the execution trace is unbounded in the past and *stationary*. We consider this as the most reasonable assumption that one can make about the past.

4.3. Examples

To illustrate our logic, we consider a few relatively standard examples in the distributed systems literature (see, e.g., [21]). The first example is *leader election* for a network of processes. The key requirement for leader election is that there is at-most one leader. Assume the number of processes is n, and `state` is a variable in each process that can have values `leader`, `loser`, `candidate`, `sleep`. We can formulate the key leader election property at every process as: "if a leader is elected then if the current process is a leader then, to its knowledge, none of the other processes is a leader" written as the PT-DTL i-local formula:

$$\texttt{leaderElected} \rightarrow \quad \begin{array}{l}(\texttt{state} = \texttt{leader} \rightarrow \\ \bigwedge_{j \neq i}(@_j(\texttt{state} \neq \texttt{leader}))\end{array}$$

Given an implementation of the leader election problem, one can monitor this formula at each process. If the property is violated, then clearly the leader election implementation is incorrect.

The second example is *majority vote*. The desired property, "if the resolution is accepted then more than half of the processes say yes", can be stated as:

$$\texttt{accepted} \rightarrow \quad \begin{array}{l}(@_1(\texttt{vote}) + @_2(\texttt{vote}) + \\ \ldots + @_n(\texttt{vote})) > n/2\end{array}$$

where, a process stores 1 in a local variable `vote` if it is in favor of the resolution, and 0 otherwise.

A third example is a safety property that a server must satisfy in case it reboots itself: "the server accepts the command to reboot only after knowing that each client is inactive and aware of the warning about pending reboot." The property is expressed as the *server*-local formula below which contains nested epistemic operators:

$$\begin{array}{l}\texttt{rebootAccepted} \rightarrow \\ \bigwedge_{\texttt{client}}(@_{\texttt{client}}(\texttt{inactive} \wedge @_{\texttt{server}}\texttt{rebootWarning}))\end{array}$$

5. Monitoring Algorithm for PT-DTL

We describe an automated technique to synthesize efficient distributed monitors for safety properties in distributed systems expressed in PT-DTL. We assume that one or more processes are associated with PT-DTL formulae that must be satisfied by the distributed computation. The synthesized monitor is *distributed*, in the sense that it consists of separate, *local monitors* running on each process. A local monitor may attach additional information to an outgoing message from the corresponding process. This information can subsequently be extracted by the monitor on the receiving side without changing the underlying semantics of the distributed program. The key guiding principles in the design of this technique are:

- A local monitor should be fast, so that monitoring can be done online;
- A local monitors should have little memory overhead, in particular, it should *not* need to store the entire history of events on a process; and
- The number of messages that need to be sent between processes for the purpose of monitoring should be minimal.

In this section, when we refer to a remote expression or formulae we mean an expression which occurs in any of the monitored PT-DTL formulae.

5.1. Knowledge Vectors

Consider the problem of evaluating a remote j-expression $@_j\xi_j$ at process p_i. A naive solution is that process p_j simply piggybacks the value of ξ_j evaluated at p_j,

$\mathcal{C}, s_i \models \text{true}$	for all s_i
$\mathcal{C}, s_i \not\models \text{false}$	for all s_i
$\mathcal{C}, s_i \models P(\xi_i, \ldots, \xi_i')$	iff $P((\mathcal{C}, s_i)[\![\xi_i]\!], \ldots, (\mathcal{C}, s_i)[\![\xi_i']\!]) = \text{true}$
$\mathcal{C}, s_i \models \neg F_i$	iff $\mathcal{C}, s_i \not\models F_i$
$\mathcal{C}, s_i \models F_i \; op \; F_i'$	iff $\mathcal{C}, s_i \models F_i$ op $\mathcal{C}, s_i \models F_i'$
$\mathcal{C}, s_i \models \odot F_i$	iff if $\exists s_i' \,.\, s_i' \lessdot s_i$ then $\mathcal{C}, s_i' \models F_i$ else $\mathcal{C}, s_i \models F_i$
$\mathcal{C}, s_i \models \Diamond F_i$	iff $\exists s_i' \,.\, s_i' \preccurlyeq s_i$ and $\mathcal{C}, s_i' \models F_i$
$\mathcal{C}, s_i \models \boxdot F_i$	iff $\mathcal{C}, s_i \models F_i$ for all $s_i' \preccurlyeq s_i$
$\mathcal{C}, s_i \models F_i \; \mathcal{S} \; F_i'$	if $\exists s_i' \,.\, s_i' \preccurlyeq s_i$ and $\mathcal{C}, s_i' \models F_i'$ and $\forall s_i'' \,.\, s_i' \prec s_i'' \preccurlyeq s_i$ implies $\mathcal{C}, s_i'' \models F_i$
$\mathcal{C}, s_i \models @_j F_j$	iff $\mathcal{C}, s_j \models F_j$ where $s_j = causal(s_i)$
$(\mathcal{C}, s_i)[\![v_i]\!]$	$= s_i(v_i)$, that is, the value of v_i in s_i
$(\mathcal{C}, s_i)[\![c_i]\!]$	$= c_i$
$(\mathcal{C}, s_i)[\![f(\xi_i, \ldots, \xi_i')]\!]$	$= f((\mathcal{C}, s_i)[\![\xi_i]\!], \ldots, (\mathcal{C}, s_i)[\![\xi_i']\!])$
$(\mathcal{C}, s_i)[\![@_j \xi_j]\!]$	$= (\mathcal{C}, s_j)[\![\xi_j]\!]$ where $s_j = causal_j(s_i)$

Table 2. Semantics of PT-DTL

with every message that it sends out. The recipient process p_i can extract this value and use it as the value of $@_j \xi_j$. However, this approach is problematic: recall that messages from p_j could reach p_i in an arbitrary order: because the arrival order of two messages, even from the same sender, is indeterminate, more recent values may be overwritten by older ones. To keep track of the causal history, or in other words the most recent knowledge, we add an event number corresponding to the local history sequence at p_j at the time expressions were sent out in messages. Stale information in a reordered message sequence is then simply discarded.

Causal ordering can be effectively accomplished by using an array called KNOWLEDGEVECTOR with an entry for any process p_j for which there is an occurrence of $@_j$ in any PT-DTL formula at any process. Note that knowledge vectors are motivated and inspired by vector clocks [8, 15]. The size of KNOWLEDGEVECTOR is not dependent on the number of processes but on the number of remote expressions and formulae. Let $KV[j]$ denote the entry for process p_j on a vector KV. $KV[j]$ contains the following fields:

- The sequence number of the last event seen at p_j, denoted by $KV[j].seq$;
- A set of values $KV[j].values$ storing the values j-expressions and j-formulae.

Each process p_i keeps a local KNOWLEDGEVECTOR denoted by KV_i. The monitor of process p_i attaches a copy of KV_i with every outgoing message m. We denote the copy by KV_m. The algorithm for the update of KNOWLEDGEVECTOR KV_i at process p_i is as follows:

1. **[internal]:** update $KV_i[i]$. Evaluate $eval(\xi_i, s_i)$ and $eval(F_i, s_i)$ (see Subsection 5.2) for each $@_i \xi_i$ and $@_i F_i$, respectively, and store them in the set $KV_i[i].values$;
2. **[send m]:** $KV_i[i].seq \leftarrow KV_i[i].seq + 1$. Send KV_i with m as KV_m;
3. **[receive m]:** for all j, if $KV_m[j].seq > KV_i[j].seq$ then $KV_i[j] \leftarrow KV_m[j]$, that is,
   $KV_i[j].seq \leftarrow KV_m[j].seq$, and
   $KV_i[j].values \leftarrow KV_m[j].values$.

We call this the KNOWLEDGEVECTOR algorithm. Informally, $KV_i[j].values$ contains the latest values that p_i has for j-expressions or j-formulae. Therefore, for the value of a remote expression or formula of the form $@_j \xi_j$ or $@_j F_j$, process p_i can just use the entry corresponding to ξ_j or F_j in the set $KV_i[j].values$. Note that the sequence number needs to be incremented only when sending messages. The correctness of the algorithm is relatively straightforward and we skip its formal proof.

Proposition 1 *For any process p_i and any j, the entry for ξ_j or F_j in $KV_i[j].values$ contains the value of $@_j \xi_j$ or $@_j F_j$, respectively.*

The above algorithm tries to minimize the local work when sending a message. However, observe that the values calculated at step 1 are needed only when an outgoing message is generated at step 2, so one could have just evaluated all the expressions ξ_i and F_i at step 2, right before the message is sent out. This would reduce the runtime overhead at step 1 but it would increase it at step 2. For different applications, different alternates may be more efficient.

The initial values for all the variables in a distributed program can may be found either by a static analysis of the program or by a distributed broadcast at the beginning of the computation. Thus, it is assumed that each process p_i has the complete knowledge of the initial values of remote expressions for all processes. These values are used to initialize the entries $KV_i[j].values$ in the KNOWLEDGEVECTOR of p_i for all j.

5.2. Monitoring a Local PT-DTL Formula

The monitoring algorithm for a PT-DTL formula is similar in spirit to that for an ordinary PT-LTL formula described in [20]. The key difference is that we allow remote expressions and remote formulae whose values and validity, respectively, need to be transferred from the remote process to the current process. Once the KNOWLEDGEVECTOR is properly updated, the local monitor can compute the boolean value of the formula to be monitored, by recursively evaluating the boolean value of each of its subformulae in the current state. To do so, it may also use the boolean values of subformulae evaluated in the previous state and the values of remote expressions and remote formulae.

A function *eval* is defined next. *eval* takes advantage of the recursive nature of the temporal operators (see Table 3) to calculate the boolean value of a formula in the current state in terms of (a) its boolean value in the previous state and (b) the boolean value of its subformulae in the current state. The function *op(F_i)* returns the operator of the formula F_i, *binary(op(F_i))* returns *true* if *op(F_i)* is binary, *unary(op(F_i))* returns *true* if *op(F_i)* is unary, *left(F_i)* returns the left subformula of F_i, *right(F_i)* returns the right subformula of F_i when *op(F_i)* is binary, and *subformula(F_i)* returns the subformula of F_i otherwise. The variable index represents the index of a subformula.

$\textbf{array}\ now;\ \textbf{array}\ pre;\ \textbf{int}\ index;$

$\textbf{boolean}\ eval(Formula\ F_i, \textbf{State}\ s_i)\{$
 $\textbf{if}\ binary(op(F_i))\ \textbf{then}\{$
  $lval \leftarrow eval(left(F_i), s_i);$
  $rval \leftarrow eval(right(F_i), s_i);\}$
 $\textbf{else if}\ unary(op(F_i))\ \textbf{then}$
  $val \leftarrow eval(subformula(F_i), s_i);$
 $index \leftarrow 0;$
 $\textbf{case}(op(F_i))\ \textbf{of}\{$
  $true : \textbf{return}\ true;\quad false : \textbf{return}\ false;$
  $P(\vec{\xi_i}) : \textbf{return}\ P(eval(\xi_i, s_i), \ldots, eval(\xi_i', s_i)));$
  $op:\ \textbf{return}\ rval\ op\ lval;\ \neg :\ \textbf{return not}\ val;$
  $\mathcal{S} : now[index] \leftarrow (pre[index]\ \textbf{and}\ lval)\ \textbf{or}\ rval;$
   $\textbf{return}\ now[index{+}{+}];$
  $\boxdot : now[index] \leftarrow pre[index]\ \textbf{and}\ val;$
   $\textbf{return}\ now[index{+}{+}];$
  $\Diamond : now[index] \leftarrow pre[index]\ \textbf{or}\ val;$
   $\textbf{return}\ now[index{+}{+}];$
  $\odot : now[index] \leftarrow val;\ \textbf{return}\ pre[index{+}{+}];$
  $@_j F_j : \textbf{return}$ value of F_j from $KV_i[j].values;$
 $\}$
$\}$

where, the global array *pre* contains the boolean values of all subformulae in the previous state that will be required in the current state, while the global array now, after the evaluation of *eval*, will contain the boolean values of all subformulae in the current state that may be required in the next state. Note that the *now* array's value is set in the function *eval*. The function *eval* on expressions is defined next:

$\textbf{value}\ eval(\textbf{Expression}\ \xi_i, \textbf{State}\ s_i)\{$
 $\textbf{case}(\xi_i)\ \textbf{of}\{$
  v_i: $\textbf{return}\ s_i(v_i);\ c_i: \textbf{return}\ c_i;$
  $f(\xi_i^1, \ldots, \xi_i^k)$: $\textbf{return}\ f(eval(\xi_i^1, s_i), \ldots, eval(\xi_i^k, s_i));$
  $@_j\xi_j'$: $\textbf{return}$ value of ξ_j' from $KV_i[j].values;$
 $\}$
$\}$

Note that the function *eval* cannot be used to evaluate the boolean value of a formula at the first event, as the recursion handles the case $n = 1$ in a different way. We define the function *init* to handle this special case as implied by the semantics of PT-DTL in Tables 2 and 3 on one event traces:

$\textbf{boolean}\ init(Formula\ F_i, \textbf{State}\ s_i)\{$

 $\textbf{if}\ binary(op(F_i))\ \textbf{then}\{$
  $lval \leftarrow init(left(F_i, s_i));$
  $rval \leftarrow init(right(F_i, s_i));\}$
 $\textbf{else if}\ unary(op(F_i))\ \textbf{then}$
  $val \leftarrow init(subformula(F_i, s_i));$
 $index \leftarrow 0;$
 $\textbf{case}(op(F_i))\ \textbf{of}\{$
  $true : \textbf{return}\ true;\quad false : \textbf{return}\ false;$
  $P(\vec{\xi_i}) : \textbf{return}\ P(eval(\xi_i, s_i), \ldots, eval(\xi_i', s_i)));$
  $op:\ \textbf{return}\ rval\ op\ lval;\ \neg :\ \textbf{return not}\ val;$
  $\mathcal{S} :\ now[index] \leftarrow rval;\ \textbf{return}\ now[index{+}{+}];$
  $\boxdot, \Diamond, \odot :\ now[index] \leftarrow val;\ \textbf{return}\ now[index{+}{+}];$
 $\}$
$\}$

As mentioned earlier, in order to properly update the set $KV_i[i].values$, we can either use the function *eval* after every internal event, or use it immediately before sending any message. If a monitored PT-DTL formula F_i is specified for a process p_i, we call p_i as the owner of that formula. At the owner process, we evaluate F_i using the $eval$ function after every internal and receive event and assign *now* to *pre*. This is done after the KNOWLEDGEVECTOR is updated, correspondingly after the event. If the evaluation of F_i is false then we report a warning that the formula F_i has been violated. The time and space complexity of this algorithm at every event is $\Theta(m)$, where m is the size of the original local formula.

5.3. Example

Consider three processes, p_1, p_2 and p_3. p_1 has a local variable x whose initial value is 5, p_2 has a local variable

$$
\begin{array}{lcl}
\mathcal{C}, s_i \models \Diamond F_i & = & \mathcal{C}, s_i \models F_i \text{ or } (\exists s'_i \,.\, s'_i \lessdot s_i \text{ and } \mathcal{C}, s'_i \models \Diamond F_i) \\
\mathcal{C}, s_i \models \boxdot F_i & = & \mathcal{C}, s_i \models F_i \text{ and } (\exists s'_i \,.\, s'_i \lessdot s_i \text{ implies } \mathcal{C}, s'_i \models \boxdot F_i) \\
\mathcal{C}, s_i \models F_i \mathcal{S} F'_i & = & \mathcal{C}, s_i \models F'_i \text{ or} \\
 & & (\mathcal{C}, s_i \models F_i \text{ and } \exists s'_i \,.\, s'_i \lessdot s_i \text{ and } \mathcal{C}, s'_i \models F_i \mathcal{S} F'_i)
\end{array}
$$

Table 3. Recursive Semantics of PT-DTL

y with initial value 7 and p_2 monitors the formula $\boxdot(y \geq @_1 x)$. An example computation is shown in Figure 3.

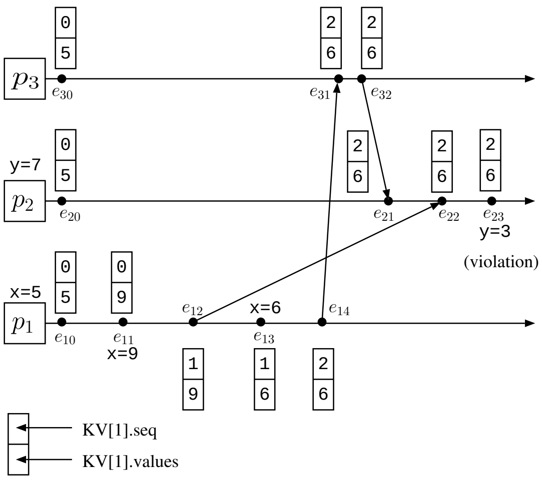

Figure 3. Monitoring of $\boxdot(y \geq @_1 x)$ at p_2

There is only one formula to monitor with a single occurrence of an @ operator, namely $@_1 x$. Hence, the KNOWLEDGEVECTOR has a single entry which corresponds to p_1. Moreover, since the only remote expression to be tracked is x, $KV[1].values$ simply stores the value of x. In the figure, next to each event, we show $KV[1]$ at that instant for that process. $KV[1]$ is graphically displayed by a stack of two numbers, the top number showing $KV[1].seq$ and the bottom number showing the value for x.

The computation starts off with the initial values of $x = 5$ and $y = 7$. All processes know the initial value of x, hence the $KV[1].values$ for each process has value 5. It is easy to see that the monitored formula $\boxdot(y \geq @_1 x)$ holds initially at p_2. Subsequently, at p_1 there is an internal event e_{11} which sets $x = 9$ and updates $KV_1[1].values$ correspondingly. Process p_1 then sends a message to p_2 with a copy of its current KV. Another internal event e_{13} causes x to be set to 6. Process p_1 again sends a message, this time to p_3, with the updated KV. Process p_3 updates its KV and sends this on the message it sends to p_2.

At process p_2, the message sent by p_3 happens to arrive earlier than the message from p_1. Therefore, at event e_{21}, on receiving the message from p_3, process p_2 is able to update its KV to the one sent at event e_{14}. The monitor at p_2 again evaluates the property and finds that it still holds. The message sent by p_1 finally arrives at e_{22} but the KV piggybacked on is ignored as it has a smaller $KV[1].seq$ than $KV_2[1].seq$. The monitor correctly continues to declare the property valid. However, another internal event at p_2 causes the value of y to drop to 3, at which point the monitor detects a property violation.

6. The DIANA Tool

We have implemented the above technique as a tool, called DIANA (DIstributed ANAlysis) (see Figure 1). DIANA is publicly available and can be downloaded from: `http://fsl.cs.uiuc.edu/diana/`. Both DIANA and the framework under which it operates are written in Java.

6.1. Actors

A number of formalisms can be used to reason about distributed systems, the most natural one being Actors [2, 3]. Actors are a model of distributed reactive objects and have a built-in notion of encapsulation and interaction, making them well suited to represent evolution and coordination between interacting components in distributed applications. Conceptually, an actor encapsulates a state, a thread of control, and a set of procedures which manipulate the state. Actors coordinate by asynchronously sending messages to each another. In the actor framework, a distributed system consists of different actors communicating through messages. Thus, there is an actor for each process in the system.

In the implementation, each type of actor (or process) is denoted by a Java class that extends a base class `Actor`. This base class implements a message queue and provides the method `send` for asynchronous message sending. Each actor object executes in a separate thread. The state of an actor is represented by the fields of the Java class. Each Java class also contains a set of `public` methods that can be invoked in response to messages received from other actors. A system level actor called *ActorManager* takes a message and transfers it to the message queue of the target actor. The target actor takes an available message from the message queue and invokes the method mentioned in the message.

While processing a message, an actor may send messages to other actors. Message sending, being asynchronous, never blocks an actor. However, an actor blocks if there is no message in its message queue. The system is initialized by the *ActorManager* object that creates all the actors in the system and starts the execution of the system.

6.2. Distributed Monitors in DIANA

The user of DIANA specifies the local PT-DTL formulae to be monitored on each actor in a special file. Each actor has a unique name, which is the name of the corresponding process. The name is passed as a string at the time creation of an actor.

As Figure 1 shows, the core of DIANA consists of two modules: an *instrumentation* module and a *monitoring* module. The instrumentation module takes the specification file and the distributed program written in the above framework and creates a Java class `MonitorImpl` that implements a local monitor for each actor (or process). It also automatically instruments the distributed program *at the bytecode level* (after compilation), so that the distributed program invokes its local monitor whenever it modifies a field variable (internal event), sends a message, or invokes a method (receive event).

One may alternately choose to evaluate the epistemic expressions and formulae immediately before sending an event (Subsection 5.1). In this case the local monitor is not invoked when field variables are modified. While runtime overhead was not the major concern for us in implementing our prototype, delaying such evaluation generally reduces the runtime overhead.

6.3. Test Cases

We implemented the following voting algorithm: a `Chair` process asks for vote on a resolution from N voters named $\texttt{Voter}_1$, $\texttt{Voter}_2$, ..., $\texttt{Voter}_N$, where N is initialized to an arbitrary but fixed positive number. We assume that the processes are connected in a tree kind of network with the `Chair` at the root of the tree and the voters at different nodes. Each voter randomly decides if it wants to vote for or against the resolution, and correspondingly stores 1 or 0 in a local state variable called `vote`. The voter then sends its decision to its immediate parent in the tree. The parent collects the votes and sends the sum of its vote and its progenies' votes to its immediate parent. The `Chair` process collects all the votes and rejects the resolution only if half or more voters have rejected. We monitor the following safety property at `Chair`:

$$\texttt{reject} \rightarrow ((\textstyle\sum_{i\in[1..N]} @_{\texttt{Voter}_i}(\texttt{vote})) < N/2)$$

The property was found to be violated in several runs: at some voter nodes, the voter sent the sum of its progenies' votes without adding its own vote. This resulted in the rejection of the resolution when it should have been accepted.

We have also tested a vector clock [8, 15] algorithm implemented in the framework presented in this section. The algorithm was implemented as part of global snapshot and garbage collection algorithm. In this algorithm, each process is assumed to have a local vector clock V that it updates according to the standard vector clock algorithm [8] whenever there is an internal event, a send event or a receive event. The safety property that the algorithm must satisfy is that, at every process p_i: "all entries of the local vector clock must be greater than or equal to the local vector clock in a causally latest preceding state of any other process," expressed as the following i-formula:

$$\boxdot(\bigwedge_{j\in[1..n]} V \geq @_j V)$$

where $V \geq V'$ when every entry in V is greater than or equal to the corresponding entry in V'. Another safety property states that "at every process p_i the i-th entry in its local vector clock must be strictly greater than the i-th entry of the local vector clock of any other process". This can be expressed as the following i-formula:

$$\boxdot(\bigwedge_{j\in[1..n]} V[i] > @_j V[i])$$

The second property was found to be violated in some computations due to a bug caused by failure to increment the i-th entry of the local vector clock of process p_i when receiving events.

These simple examples illustrate the practical utility and power of PT-DTL and the monitoring tool DIANA based on it.

7. Conclusion and Future Work

This work represents the first step in effective distributed monitoring. The work presented here suggests a number of problems that require further research. The logic itself could be made more expressive so that it expresses not only safety, but also liveness properties. One difficulty is that software developers are reluctant to use formal notations. A partial solution may be to merge the present work with a more expressive and programmer friendly monitoring temporal logic such as EAGLE [6]. A complementary approach is to develop visual notations and synthesizing temporal logic formulas from such notations. There may also be the possibility of learning formulas based on representative scenarios.

An interesting avenue of future investigation that our work suggests is what we call *Knowledge-based*

Aspect-Oriented Programming. Knowledge-based Aspect-Oriented Programming is a meta-programming discipline that is suitable for distributed applications. In this programming paradigm, appropriate actions are associated with each safety formula; these actions are taken whenever the formula is violated to guide the program and avoid catastrophic failures.

Acknowledgements

The first three authors are supported in part by the Defense Advanced Research Projects Agency (the DARPA IPTO TASK Program, contract F30602-00-2-0586, the DARPA IXO NEST Program, contract F33615-01-C-1907), the ONR Grant N00014-02-1-0715, and the Motorola Grant MOTOROLA RPS #23 ANT. The last author is supported in part by the joint NSF/NASA grant CCR-0234524.

References

[1] *1st, 2nd and 3rd CAV Workshops on Runtime Verification (RV'01 - RV'03)*, volume 55(2), 70(4), 89(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science: 2001, 2002, 2003.

[2] G. Agha. *Actors: A Model of Concurrent Computation*. MIT Press, 1986.

[3] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.

[4] R. Alur, D. Peled, and W. Penczek. Model checking of causality properties. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS'95)*, pages 90–100, San Diego, California, 1995.

[5] R. Aumann. Agreeing to disagree. *Annals of Statistics*, 4(6):1236–1239, 1976.

[6] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proceedings of 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57, Venice, Italy, January 2004. Springer-Verlag.

[7] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning about Knowledge*. MIT Press, 1995.

[8] C. J. Fidge. Partial orders for parallel debugging. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging (WPDD'88)*, pages 183–194. ACM, 1988.

[9] K. Havelund and G. Roşu. Java pathexplorer – A runtime verification tool. In *The 6th International Symposium on Artificial Intelligence, Robotics and Automation in Space: A New Space Odyssey*, Montreal, Canada, June 18 - 21, 2001.

[10] K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer-Verlag, 2002.

[11] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-mac: a run-time assurance tool for java. In *Proceedings of Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.

[12] M. Leucker. Logics for mazurkiewicz traces. Technical Report AIB-2002-10, RWTH, Aachen, Germany, April 2002.

[13] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, 1992.

[14] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.

[15] F. Mattern. Virtual time and global states of distributed systems. In M. C. et. al., editor, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science, 1989.

[16] B. Meenakshi and R. Ramanujam. Reasoning about message passing in finite state environments. In *International Colloquium on Automata, Languages and Programming (ICALP'00)*, volume 1853 of *Lecture Notes in Computer Science*, pages 487–498. Springer-Verlag, 2000.

[17] W. Penczek. A temporal approach to causal knowledge. *Logic Journal of the IGPL*, 8(1):87–99, 2000.

[18] W. Penczek and S. Ambroszkiewicz. Model checking of causal knowledge formulas. In *Workshop on Distributed Systems (WDS'99)*, volume 28 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1999.

[19] R. Ramanujam. Local knowledge assertions in a changing world. In *Theoretical Aspects of Rationality and Knowledge (TARK'96)*, pages 1–14. Morgan Kaufmann, 1996.

[20] K. Sen, G. Roşu, and G. Agha. Runtime safety analysis of multithreaded programs. In *ACM SIGSOFT Conference on the Foundations of Software Engineering / European Software Engineering Conference (FSE / ESEC'03)*, Helsinki, Finland, 2003.

[21] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2nd edition, September 2000.

[22] P. S. Thiagarajan and I. Walukiewicz. An expressively complete linear time temporal logic for Mazurkiewicz traces. In *Twelth Annual IEEE Symposium on Logic in Computer Science (LICS'97)*, pages 183–194, Warsaw, Poland, 1997.